Fast Collision Culling in Large-Scale Environments Using GPU Mapping Function

Quentin Avril, Valérie Gouranton, Bruno Arnaldi

Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs and T. Kuhlen (Editors)

Fast Collision Culling in Large-Scale Environments Using GPU Mapping Function

Q. Avril[1] and V. Gouranton[2] and B. Arnaldi[2]

[1]University of Rennes 1
[2]INSA of Rennes

Abstract

This paper presents a novel and efficient GPU-based parallel algorithm to cull non-colliding object pairs in very large-scale dynamic simulations. It allows to cull objects in less than 25ms with more than 100K objects. It is designed for many-core GPU and fully exploits multi-threaded capabilities and data-parallelism. In order to take advantage of the high number of cores, a new mapping function is defined that enables GPU threads to determine the objects pair to compute without any global memory access. These new optimized GPU kernel functions use the thread indexes and turn them into a unique pair of objects to test. A square root approximation technique is used based on Newton's estimation, enabling the threads to only perform a few atomic operations. A first characterization of the approximation errors is presented, enabling the fixing of incorrect computations. The I/O GPU streams are optimized using binary masks. The implementation and evaluation is made on large-scale dynamic rigid body simulations. The increase in speed is highlighted over other recently proposed CPU and GPU-based techniques. The comparison shows that our system is, in most cases, faster than previous approaches.

1. Introduction

Collision detection is a well-studied and active research field where the main problem is to determine how and if one or more objects collide in a 3D virtual environment. Collision detection is an issue affecting many different fields of study, including physical-based simulation, computer animation, robotics, haptic applications and video games. In these applications, real-time performance, efficiency and robustness are key issues. In the field of Virtual Reality, physical virtual environments in digital mock-ups and industrial applications are now commonplace, and are of increasing complexity. The expected level of real time performance is becoming harder to ensure in such large-scale virtual environments. Unsurprisingly, collision detection has been an integral part of virtual reality bottlenecks for over thirty years.

The use of parallel processing has become necessary to take advantage of recent gains in Moore's Law. Over the past several years, processor specialists have provided clock frequency increases and parallelism improvements in instruction sets. Now, to better manage power consumption, they promote multi-core architectures. It is no longer possible to rely on the evolution of processing power to overcome the problem of real-time collision detection. The impressive evolution of graphics hardware and multi-GPU platforms have provided programmers with better processing tools. With these new tools it is now essential to take into account

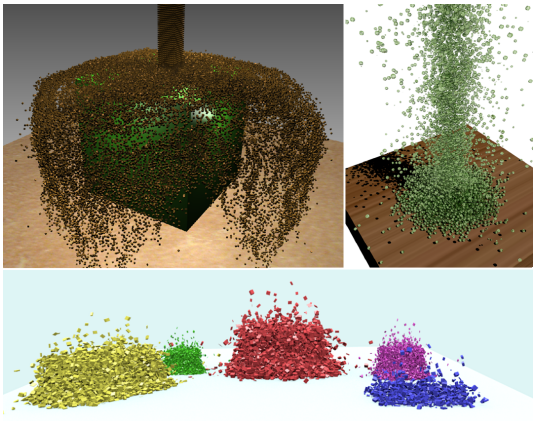


Figure 1: *250K cubes, 100K spheres or 75K rectangles. Using the GPU mapping function, culling non-colliding pairs is performed in less than 25ms on a Nvidia Quadro FX3600M in the spheres environment.*

run-time architectures to improve collision detection performance in large-scale virtual environments.

Main Results: A fast and novel approach to cull non-colliding object pairs in large-scale environments is proposed. The approach is based on the use of new semi brute-force GPU mapping function used during the broad phase step. New access functions are proposed to optimize execution time of the GPU kernel reducing global GPU memory accesses. The kernel time is also reduced using square root approximation based on Newton's approximation where a first way to characterize and fix errors is presented. Input and output data transfers are optimized using binary masks to fully use the entire binary stream. This new broad phase algorithm enables to get an "interactive" computation time (<25ms) with more than 100K objects.

Organization: The rest of the paper is organized as follows: in Section 2 related work on parallel collision detection and broad phase techniques is reported. Section 3 describes the new brute force broad phase algorithm and Section 4 gives details on the dynamic spatial subdivision technique adapted to a GPU use. The implementation and performance measurements are described in Section 5. The comparison and analysis, over major CPU and GPU-based techniques, are done in Section 6. Then, the conclusion and the opening on future work end the paper.

2. Related Work

A brief overview is presented on parallel solutions for collision detection focusing on the broad phase step. For more details on the collision detection field, the reader may refer to surveys on the topic [LG98, TKH*05].

2.1. Broad Phase Collision Detection

The collision detection process is organized and built as a pipeline [Hub95] composed by two main parts: broad and narrow phase. The broad phase step is in charge of a quick and efficient removal of object pairs that are not in collision. In the bounding volume family many models have been proposed such as spheres [Hub95], Axis-Aligned-Bounding-Box (AABB) [Ber97], Oriented-Bounding-Box (OBB) [GLM96], discrete oriented polytopes (k-DOP) and many others. Other methods have been proposed to use spatial partitioning and divide space into unit cells: regular grid [Ove92] or non-regular [Mir97, EG07], quadtree, octree [BT95], Binary Space Partitioning (BSP) or k-d tree structure. Topological approach is based on the positions of objects (*Sweep and Prune* (SaP) [CLMP95]). There are two related but different concepts on the way the SaP operates internally: starting from scratch each time (brute force) or updating internal structures (persistent). The persistent algorithm has been enhanced by using a segmented interval list combined with subdivision [TBW09]. It provides faster sequential execution in large-scale environments. The brute-force SaP has been paralyzed on multi-core architecture [AGA11] and it was shown that, according to the number of objects and the number of CPU cores, the parallelization of the brute force method may become more efficient than the persistent one. Luque et al. [LCF05] proposed a new structure called *"Semi-Adjusting BSP-tree"* to represent scenes consisting of thousands of objects. They showed that the tree does not require a complete restructuring even for highly dynamic scenes.

2.2. Parallel Solution for Collision Detection

The parallel solution of collision detection algorithms is a recently explored and active field in high performance computing. There are three different families of algorithms, namely CPU, GPU and hybrid-based solutions.

A new task splitting approach for implicit time integration and collision handling on multi-core architecture has been proposed [TPB08]. Tang et al. [TMT09] propose to use a hierarchical representation to accelerate collision detection queries and an incremental algorithm exploiting temporal coherence. The overall is distributed among multiple cores. They obtained a 4X-6X speed-up on an 8-core processor based on several deformable models. Kim et al. [KHeY08] propose to use a feature-based bounding volume hierarchy (BVH) to improve performances of continuous collision detection. They also propose a new method of task decomposition for their BVH-based collision detection and dynamic task assignment methods. They obtained a 7X-8X speed-up using an 8-core architecture compared to a single-core. Hermann et al. [HRF09] propose a parallelization of interactive physical simulations. They obtain a 14X-16X speed-up on a 16-core architecture compared to a single-core. Tang et al. [TLW11] proposed a new parallel approach based on a new concept of filters into subspaces. They propose a new phase between the broad and narrow phase where they apply two successive filters (linear and planar filters). This technique enables to widely prune elementary tests and significantly reduce the computation time.

Cinder [KP03] is an algorithm exploiting the GPU to implement a ray-casting method to detect collision. GPU-based algorithms for self-collision and cloth animation have also been introduced by Govindaraju et al. [GLM05]. A solution using image-space visibility queries has been proposed for the broad phase [GRLM03]. Le Grand [LG07] has presented a GPU implementation of the SaP broad phase algorithm based on CUDA faster than the CPU implementation. He described how to implement a spatial subdivision method for GPU between particles. Liu et al. [LHLK10] recently presented a new GPU-based SaP. They propose to reduce the density of intervals along the sweep axis by using principal component analysis to choose the best sweep direction. They couple it with spatial subdivisions to further reduce the number of false positive overlaps. Recent works use thread and data parallelism on a single GPU to perform fast hierar-

chy construction, updating, and traversal using tight-fitting bounding volumes such as oriented bounding boxes (OBB) and rectangular swept spheres (RSS) [LMM10]. Tang et al. [TMLT11] proposed a new GPU-based streaming method to reduce access time and computation for the calculation of intersection between primitives.

Kim and al. [KHH*09] presented a hybrid parallel continuous method (*HPCCD*) based on a bounding volume hierarchy. Recently, Pabst and al. [PKS10] have presented a new hybrid CPU/GPU method for rigid and deformable objects based on spatial subdivision. The broad and narrow phases are both executed on multi-GPU architecture.

The total number of objects in recent virtual environments and games increase and become more dynamic. The performance of the most-used algorithms for the broad phase step is becoming a serious issue. The Sweep and Prune algorithm does not scale very well and its performance decreases in very large scale environments. More precisely, the insertion and sorting of new objects as well as the interaction with far away objects is very time-consuming.

3. Optimized Broad Phase

In a n-body simulation, a semi-brute force algorithm performs $\frac{n^2-n}{2}$ computations at each loop (cf. Table 1). The pairs that look like a (x,x) form are not tested (Deletion of the matrix diagonal). The symmetrical pairs are only computed once , i.e. (a,b) is tested but (b,a) is not. It is therefore a right upper triangular matrix. The computation made on each pair consists in determining if two bounding volumes overlap or not. As AABBs are used, each object has 6 values: its minimal and maximal positions on the 3 axis. To detect overlaps, an algorithm is used to determine if one of the maximum borders of an object is inferior to the minimum border of the other object on the same axis. If so, it means that there is a $max - min$ space between both bounding volumes and there is no collision.

3.1. The GPU Mapping Function

In order to parallelize computations on GPU, each thread is in charge of one object pair. For a n-body simulation, $\frac{n^2-n}{2}$ threads will be created. It is necessary to supply in every thread the pair of objects which it has to compute. Each object consists in 2 vectors (min and max) and each of them consist in three values (x, y *and* z) so 12 values must be transmitted to each of the threads. The accuracy of these values is essential to ensure a consistent result for the presence or absence of a collision, it is thus essential that each of these 12 values must be coded on 4 bytes (floating point number). $12*4 = 48$ bytes have to be transmitted to each thread. With one or ten thousand of objects, the transfer size would be respectively 23MB or 2.3GB. It is quite obvious that the transfer size quickly grows not linearly.

Each GPU thread has its own unique *ID*, the thread 0 deals with the pair 0 until the thread $\frac{n^2-n}{2}$ dealing with the $(\frac{n^2-n}{2})^{th}$ pair. As the CPU-GPU transfer occurs at each time step, the data should not be as big as shown previously. The main goal is to minimize the transfer size while preserving the quality of the information. In the previous example, each object was transmitted $n-1$ times to the GPU at each time step, it is possible to consider sending each object only once per step that would significantly reduce the transfer size. For a simulation with 1K objects, the size would be: 23KB.

But it is no longer possible for a thread to know which pair it has to compute. The first idea would be to allocate $\frac{n^2-n}{2}$ locations in the GPU memory. They would correspond to the pairs' *ID* that each thread has to compute. Working with simulations with a fixed number of objects, it is possible to create and to store the list of the pairs' identifiers during an initialization phase. For example, let *pairs_memory* be the allocated memory containing pairs of objects' *ID*. The thread k accesses on the k^{th} location of *pairs_memory* and reads two values $object1_{ID}$ and $object2_{ID}$. These two *IDs* are then used to localize the objects in the global memory in order to get their bounding volume's coordinates. In this case, each thread does two global memory accesses to determine which objects it has to compute, followed by two other global memory accesses to get bounding volume's coordinates. Each thread performs four memory accesses before performing its computations.

To reduce the number of memory accesses, a novel approach is proposed to determine the pair (a,b) that a thread has to compute without any memory access only with the number of objects n and the identifier of the thread k.

Example: In a simulation with 1000 objects, the pairs to handle are: (1,2),(1,3),(1,4), ... , (998,1000),(999,1000). The goal is to find a method to determine, without memory access, that the thread 71452 has to handle the pair (75,303) and the thread 46108, the pair (48,285).

	1	2	3	...	n-1	n
1	X	0	1	...	$n-3$	$n-2$
2	X	X	n-1	...	$2n-5$	$2n-4$
3	X	X	X	...	$3n-8$	$3n-7$
...	...	...	...	...	...	...
n-1	X	X	X	...	X	$\frac{n^2-n}{2}$
n	X	X	X	...	X	X

Table 1: *The pairs matrix in a simulation of n objects.*

3.2. Linear Function of Access

As per Table 1, it is possible to determine the objects identifiers for a given pair number, using computation loops based on mathematical series. But more the identifier of the pair is high and more the number of computation loops will be important. If each GPU kernel function has a different computation time due to an evolutionary number of loops, the

global computation time will be impacted by the slowest thread. The computation time in GPU kernels has to be homogeneous regardless of the pair identifier to handle.

3.2.1. Turn the Index Into Unique Pair

The numbering of the matrix rows and columns is done from 1 to n (from the top left to the bottom left). Let a be the index of the row and b the column one. In this case, the first line contains $(n-1)$ numbers, the second one $(n-2)$, ..., the k^{th} $(n-k)$,... and the n^{th} 0. From line 1 to line $a-1$ there is:

$$S_a = (n-1)+(n-2)+...+(n-(a-1)) \text{ elements}$$

As the pairs numbering starts at 0, S_a is the first number of the line a $(0, n-1, ..., \frac{n^2-n}{2}$ on Table 1). S_a is a series, the operation can be simplified:

$$S_a = \frac{(a-1)(2n-a)}{2}$$

As this first number is in the column $a+1$, the number that is in the column b is thus $S_a + b - (a+1)$. With the identifier k of the thread and the number of objects n, the a and b values that correspond to objects to compute have to be determined. The inequality to solve is the following:

$$S_a \leq k \leq S_{a+1}$$

$$\frac{(a-1)(2n-a)}{2} \leq k \leq \frac{((a+1)-1)(2n-(a+1))}{2}$$

But, to solve $S_a \leq k \leq S_{a+1}$, it is not necessary to use both inequalities because it is enough to say that:

- a is the larger integer such as $S_a \leq k$
- or that a is the smaller integer such as $k \leq S_{a+1}$

Which, in both cases, leads to seek for the real solution of the equation and then to take the integer part. Starting from the special case where the number of the thread corresponds to the beginning of one line of the matrix ($k = S_n$):

$$k = \frac{(a-1)(2n-a)}{2} \rightarrow -a^2 + a(2n+1) - 2n - 2k = 0$$

This second degree polynomial equation has two roots:

$$X_1 = \frac{-(2n+1)+\sqrt{\Delta}}{-2} \qquad X_2 = \frac{-(2n+1)-\sqrt{\Delta}}{-2}$$

with $\Delta = 4n^2 - 4n - 8k + 1$.

During the roots resolution, only X_1 is located between the borders 1 and n. X_2 is out of bounds of the solution and leads to a wrong coordinate in the matrix (up to the number of objects), which has no meaning. X_1 needs to be solved in order to get the y coordinate of the matrix. The integer part of the root X_1 enables to get the matrix coordinate.

$$a = \lfloor \frac{-(2n+1)+\sqrt{4n^2-4n-8k+1}}{-2} \rfloor \quad (1)$$

To get b, the following equation needs to be solved:

$$b = (a+1)+k-S_a = (a+1)+k-\frac{(a-1)(2n-a)}{2} \quad (2)$$

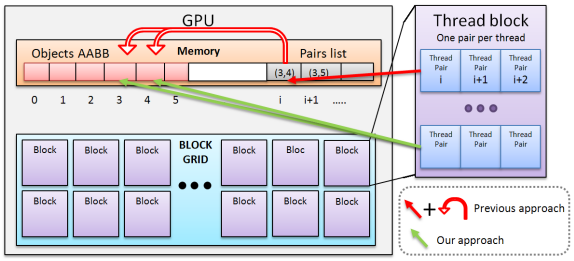


Figure 2: *General overview of our approach that reduces the GPU global memory accesses done by the threads.*

where S_a is the first value of line a. With this polynomial resolution, it is now possible for each thread to determine (using n (number of objects) and its index k) the two objects it has to deal. It is no longer necessary to allocate the pairs in the GPU memory, which saves four memory accesses per thread and a fairly substantial area of memory. The only data to transfer at each time step is the two *Min* and *Max* vectors of each object. Which, for a n objects simulation is $n*2*3*4 = 24*n$ *bytes*. With 10K objects, the size of data to transfer is 234Kb and the transfer can be performed in less than 50 μs (0.050 ms) on recent GPU architectures. This time is almost negligible compared to our problem.

3.2.2. Comparison with Lazy Way

It could have been possible to use a simpler way to retrieve the pairs with the IDs that consists in generating pairs without computing any square roots and checking if they are valid. The algorithm is for a given ID k and N objects, $a = int(k/N)$ and $b = k\%N$. Then the result is tested before handing this thread ID number back: if $a \geq b$, then this is an invalid ID that has generated an already existing pair. The calculations are stopped for this thread. While if $a < b$, then the pair is valid and the calculation on the potential overlap of objects is pursued. The point is, for a large N, about half of the IDs are thrown away in the 0 to $\frac{n^2-n}{2}$ list of numbers and pass the other half on. It is therefore necessary to test $n^2 - n$ IDs at the cost of an additional "if", integer divide and modulo operation. When comparing this approach on sequential CPU with the previously detailed method, it quickly appears that, in a sequential CPU context, it is better suited than the previous approach. In spite of the twice higher number of loops, the atomic functions are simpler and without computing the square root, the algorithm is four times faster when running both on sequential CPU. But with 10K objects, it takes 4.51s to compute on CPU... Thus the parallel optimization becomes necessary. When comparing both techniques on GPU, the results are inversed. The Table 2 shows results obtained on a Nvidia Quadro FX 3600M. Performing twice as more computations in parallel on GPU, even if the atomic operations are simpler, may be less efficient. That shows the inconsistency of the lazy way in a parallel context.

Nb objects	With sqrt (ms)	Without (ms)
1K	**0.21**	0.25
10K	**2.25**	4.28
100K	**29.66**	58.24

Table 2: *The comparison of our approach and the use of no square root to determine the objects pair with an ID.*

3.3. Square Root Optimization

Computing the square root of a floating point number is essential in many 3D applications. The new formulas used by the threads that have been proposed use square roots. A brief overview on the study on different possible optimizations for the square root computation is presented. Square root functions are very time consuming on most CPU even with particular instructions. That is why the GPU have dedicated hardware to perform such functions faster. But it still remains an important time consuming operation even on GPU. In 2005, *id Software*(www.idsoftware.com) released the source code of the Quake III engine under GPL license. In this code, two methods were found by engineers and assigned to the programmer John Carmack (Few investigations showed that the code had older roots in the hardware and software side of the computer graphics community). One of these functions is designed to compute the square root of a floating point number without any loop, only using linear elementary computations. This function is based on a very rapid Newton's-method-based approximation to the square root. The C code was essentially (without original comments):

```
float SquareRootFloat(float nb) {
    float nb_half = nb * 0.5F;
    float y = nb;
    long i = * (long *) &y;
    i = 0x5f3759df - (i >> 1);
    y = * (float *) &i;
    //Repetitions increase accuracy(6)
    y = y * (1.5f - (nb_half * y * y));
    return number * y;}
```

This method does not contain any loop thanks to the use of the "magic number" 0x5f3759df. This hexadecimal constant is used as a first approximation and significantly reduces the iterations number needed to obtain a satisfactory approximation. This code is based on the fact that the IEEE 754-2008 floating point format approximates base-2 logarithm. The Carmack's constant was then studied and improved, first by Chris Lomont [Lom03] followed by Charles McEniry [McE07]. Lomont has proposed a new constant 0x5f375a86 which appears to perform slightly better than the previous one. McEniry obtained the same constant that Lomont determined but he has proved that a constant R can be used to approximate the integer value of the inverse square root of a floating point number but he has not proved that this constant R assumes the value used in the code itself.

Lomont and McEniry reported that this square root approximation can lead to errors. As shown previously, in both formulas 1 and 2, the square root is only used with integers and the decimal part of the value is useless. In preliminary tests it has been found that for very high values of n (number of objects), errors persist and lead in our case to a wrong result. An error means that two threads with a consecutive identifier will compute the same pair which means that one pair will not be processed. To avoid getting errors in the computation of the pairs of objects, we analyzed them and successfully manage to characterize them. The square root in the GPU kernel function is $\sqrt{4n^2-4n+8k+1}$ with n, the number of objects in the simulation and k, the thread identifier. Errors can only occur in only two different situations of bad approximation of the square root. These two situations give a wrong pair (a,b) that looks like:

- $b > Nb\_obj$
- or $a > b$

In both cases, there is a way to fix it. In the first case, the rule to apply to get the exact pairs coordinates is:

```
if(b > Nb_obj){
    a++;
    b = a+(b-Nb_obj);
```

Whereas in the second case, the rule to apply is:

```
if(a >= b){
    b = nb_obj - (a - b);
    a--;}
```

The next Table 3 shows the % of errors of the Carmack's and Lomont's approximation without correcting errors. Two different tests were performed with respectively two and three accuracy steps. An error occurs when the lower integral part of the computed square root and the approximated square root are different. The percentage of error with 4 accuracy steps is exactly the same than the 3 step's one.

	2 accuracy steps		3 accuracy steps	
Nb Tests	Carmack	Lomont	Carmack	Lomont
1K	2.9%	2.9%	0.7%	0.7%
10K	0.89%	0.89%	0.12%	0.12%
100K	0.27%	0.27%	0.035%	0.035%
1M	0.1800%	**0.1803%**	0.0110%	**0.0111%**
10M	0.3681%	**0.3685%**	0.0040%	**0.0057%**
100M	0.9847%	**0.9855%**	0.0134%	**0.0142%**

Table 3: *Percentage of errors of the Carmack's and Lomont's square root approximation when using **two** or **three** accuracy steps without correcting errors.*

From 1K to 100M tests, a 0% of error is obtained, when applying the conditions (cf. above) after the computation of the square root approximation to fix the errors leading to bad pairs. It is now possible to use the method of the approximation of the square root in the kernel function being sure of the consistency of the result.

The Table 4 presents the results obtained by comparing normal CUDA square root (*sqrt()*) with the Carmack's and Lomont's approximation techniques.

Nb tests	Carmack	Lomont	CUDA	γ
1M	**0.042**ms	**0.042**ms	0.048ms	12.6%
10M	**0.409**ms	**0.409**ms	0.478ms	14.7%
100M	**4.102**ms	**4.102**ms	4.722ms	13.2%
1B	**41.01**ms	**41.01**ms	48.01ms	14.6%

Table 4: *GPU Computation time (in ms) of the Carmack's and Lomont's approximation compared to the CUDA square root function. The tests were made from 1 million to 1 billion sqrts to compute. The value in bold is the shortest computation time. The γ value correspond to the % of computation time reduction obtained with the square root approximation.*

The Carmack's and Lomont's square root approximations outperform the standard CUDA sqrt. The difference is not as significant as expected but is still a time reduction. On average, the percentage of obtained computation time is 13.7%.

3.4. Output Transfer Optimization

It is also essential to optimize as much as possible the output transfer time (from the GPU to the CPU). This step is made at each time step so shorter it is and better it is. Each thread is in charge of determining if two bounding volumes of objects are in collision or not. The answer "yes" or "no" can be represented with only one bit (1 or 0). For n objects in the simulation, there are $\frac{n^2-n}{2}$ pairs to compute, as many bits have just to be allocated in the memory. If the bit number X is 1, it means that the bounding volumes of the objects of the pair number X are in collision. In the CUDA/C++ programming context, it is not possible to directly manipulate 1 bit parameters or attributes. Even a character or a Boolean value is coded on 1 byte (8 bits). Allocate a character per pair would be 8 times more important than just one bit and the transfer time would also be longer.

The 8 bits of one character are used to store the collision test result of 8 pairs. The GPU memory allocation can only be made in bytes so the number of needed bytes is $nb\_Bytes = \lceil N/8 \rceil$. Thus, (N/8 + (1 or 0)) bytes are allocated to store results. When the collision test is positive, the thread creates a character equal to 1 and then proceeds to a binary shift of $(ID\%8)$ bits into the character (ID is the thread identifier). Then it has to write this character in the $(ID/8)^{th}$ memory location (previously allocated). But 8 threads will write to the same memory location so the result information has to be maintained in the same character. To do so, before writing on the memory, each thread has to read the $(ID/8)^{th}$ memory location and apply an *OR* logical function of the two binary strings. This elementary operation enables to do the union of the results from the 8 threads in the same character.

The read of this binary string transmitted to the CPU is made with the other logical operation *AND*. If the user wants to know if the 19^{th} object pair contains a collision, he creates a character equal to 1, then he performs three binary shifts $(19\%8 = 3)$ and applies the mask on the third character in the memory ($19/8 = 2$ but it starts at 0). If the result is more or equal to one, the bounding volumes of the objects pair $(2*8)+3 = 19$ are in collision.

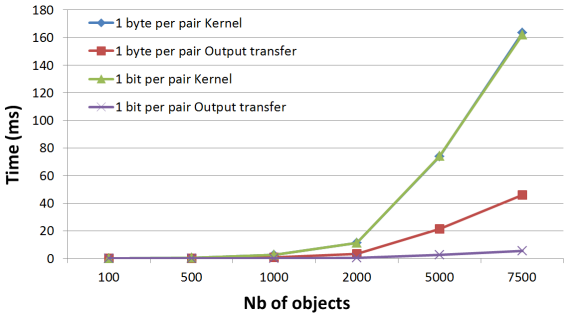


Figure 3: *The output transfer and kernel times between using 1-byte or 1-bit per objects pair. Time was measured in milliseconds from 100 to 7500 objects.*

But if 8 threads write at the same memory location, they cannot write it in parallel. These accesses are serialized. The differences between 8 parallel writings in the global GPU memory (using a 1-byte data type) and 8 serialized writings (using a 1-bit data type) were evaluated. Table 5 presents the results of these performance measurements from 100 to 7500 objects in the simulation. The kernel execution time and the output transfer time were measured. The input transfer was not controlled because the choice between 1-bit or 1-byte does not impact it. The Figure 3 shows the difference between both configurations. The kernel execution time is exactly the same between using 1-byte or 1-bit so between parallel or serialized accesses on the global memory of the GPU. While the output transfer time is significantly longer when using 1-byte per pair.

	1 byte per pair		1 bit per pair	
Nb obj	Kernel(ms)	Output(μs)	Kernel	Output
100	0.03	14.35	0.03	7.75
1000	2.70	842.75	2.72	111.18
5000	73.95	21428	74.14	2612.42
7500	163.60	45865.94	162.03	5687.20

Table 5: *Kernel and output with 1-byte or 1-bit per pair.*

3.5. Intermediate Results

Intermediate results obtained by the new broad phase algorithm based on a semi-brute force approach are presented. The graph shown on Figure 4 summarizes the evolution of the GPU kernel time according to the number of objects and the changes in the input/output data size. These results show

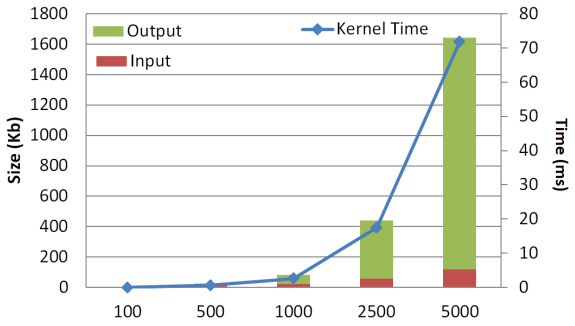


Figure 4: *GPU Kernel time, input and output data size of our model from 100 to 5000 objects.*

that the transfer time is negligible in relation to the computation time of the threads within the kernel. Data transfer from CPU to GPU increases linearly in relation to the number of objects. The output one (from the GPU to the CPU) is more important.

3.5.1. Intermediate Comparison

For the simulations with less than 3000 objects, the new model outperforms the latest approaches. If we compare the simulation results from 2500 objects with Liu et al. [LHLK10], our approach is 20% faster. Their approach is based on a standard GPU SaP coupled with an algorithm to choose the best projection axis. Compared with the approach of Pabst et al. [PKS10] and Kim et al. [KHH*09] that test their hybrid model with the *n*-body benchmark of the UNC (http ://gamma.cs.unc.edu/), our broad phase algorithm is, respectively, 5 and 3 times faster. Kim's approach is based on a tree parallel traversal and the Pabst's one is based on a uniform spatial subdivision coupled with a non-optimized for triangles brute force SaP.

3.5.2. Intermediate Synthesis

The initial objective was to show that a very simple algorithm (brute force) properly optimized, could be very powerful. It is partly achieved. The new proposed algorithm is more efficient than existing approaches for scenes less than 3000 objects but as expected, given the combinatorial nature of the algorithm with a quadratic complexity in $O(n^2)$, the results collapse when the number of objects increases. The objective is clearly the simulation of large-scale virtual environments, so it is necessary to revise this model to add conditions to prune the number of pairs to test. To do so this combinatorial effect is removed by coupling this approach with a spatial subdivision technique. The optimizations on the size and transfer time and the GPU mapping function that enables the threads to compute their current pair without any memory access are still used with the spatial subdivision.

4. Spatial Subdivision

To associate the broad phase algorithm with a spatial subdivision has become necessary to continue the reduction of the computation time by avoiding useless calculations of overlapping [LHLK10]. To further reduce the computation time and break the quadratic complexity, the new semi-brute force approach is associated to spatial subdivision. Indeed, a 3D grid is proposed to reduce the number of pairs to compute and to locally perform our brute force approach. This approach can be compared to the Multi-SAP concept, first proposed and thought by Erwin Coumans and Pierre Terdiman [Ter07] but still marked as experimental. It has been previously shown that the strategy which consists only in applying the brute-force method was adapted well to "small environments" but totally unsuitable for larger environments. It is therefore useful to widely prune the number of pairs. The grid-type structure is perfectly suited to this type of problem. This grid is set at the beginning of the simulation and then updated at each time step of the simulation. The number of cells and their size depend on the number of objects and their size. In the following, the establishment and the update of the grid structure are presented. The construction of the grid is done in two steps: (1) determining number and size of the cells and (2) putting objects in the grid cells.

4.1. Cells Characteristics

To ensure uniformity and consistency, the density of objects within the environment is considered in order to determine the characteristics of the grid cells. The dimensions of the environment and the number of objects are used in the construction. First, the objects density is evaluated in the environment, dividing the number of objects by the cubic volume of the environment. Then, a X value is set in order to determine the ideal volume that a cell should have to contain X objects. X is divided by the density of the environment for the optimal volume of cell. Then the cubic root of this volume is computed in order to determine the size of the side of a cell. In this approach the grid has cubic cells. After obtaining the size of a cell, the number of cells per axis can be determined allowing refining the cell size for a perfect fit to the size of the environment. The value X varies according to simulations.

4.2. Grid Fill in

The goal is then to link the objects to their respective cells. To do it the GPU is used to massively parallelize the computations. Each calculation has to determine in which grid cell is an object. The computation for each object is totally independent of the other computations, they can be performed in parallel on the GPU. The cells to which belong objects are determined. The cells' identifiers of one object are computed using the position of the object and the maximum and minimum bounds of it in order to loop through all relevant cells

and mark these as occupied by that object. At the end of this step, each grid cell is composed of a list of objects' identifiers. The semi-brute force algorithm presented previously is then locally applied according to the cells of the grid. The occupancy of the GPU is maximal to simultaneously run a maximum of cells in parallel.

4.3. Grid Update

This update step is essential when using a grid-like structure. The goal of the grid being roughly to prune candidate pairs to compute during the brute force algorithm, the update of its cells must therefore be fast. This step is also performed on the GPU. The GPU kernel function computes the cell identifier of each object in parallel. It thus takes into account their new positions and assigns them to the corresponding cells.

5. Implementation and Performance

In this section, the performance measurements of the new parallel broad phase algorithm is presented. In all tests, all objects are moving in order to evaluate the performance in the most extreme situations. The algorithm was developed an tested on a Intel Core-Duo CPU X7900 of 2,8 GHz with 3 Gb of memory with a Nvidia Quadro FX 3600M. The operating system is Windows XP 32 bits.

Nb Objects	α(ms)	β(ms)	δ(ms)	Total
1K	0.03	0.13	0.09	**0.25ms**
10K	0.31	1.45	0.59	**2.35ms**
100K	3.38	11.12	13.41	**27.91ms**
1M	14.26	134.91	142.72	**291.89ms**

Table 6: *Computation time - α: AABB Update - β: GPU Grid Update - δ: GPU Brute Force SAP*

Nb Objects	Without Grid	With Grid
100	**0.046ms**	1.020ms
1K	**0.872ms**	1.840ms
10K	31.450ms	**3.257ms**

Table 7: *Computation time with/without the grid use.*

The second column of the Table 6 corresponds to the step of the update of the bounding volumes (here AABBs). The third column is the time spent to update the grid on the GPU. The fourth column gives the computation time of the brute force Sweep and Prune on GPU. This time includes the time of the input and output transfers. The Table 7 shows a comparison of our approach with or without the use of the 3D grid. The use of the grid is useless in "small" environments, the execution time is better when not using it. These small environments are those composed of less than 1000 objects. Beyond these 1000 objects, using a 3D grid leads to obtain more efficient results.

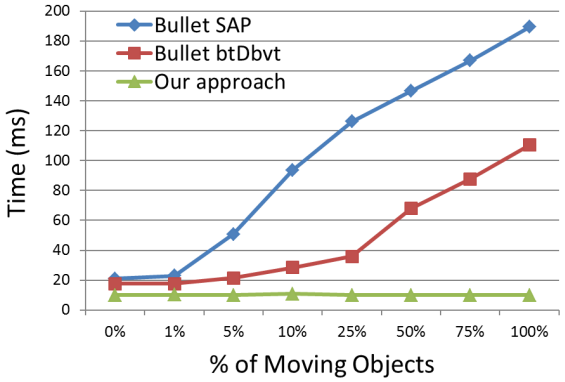


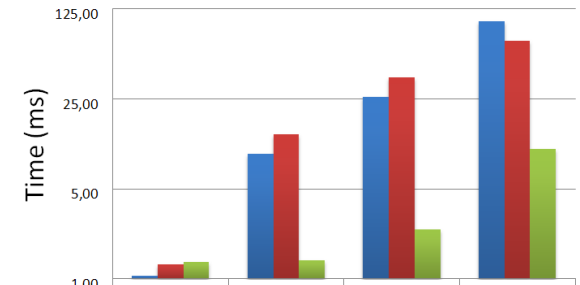
Figure 6: *Evaluation of the objects movement's impact on the computation time.*

6. Comparison and Analysis

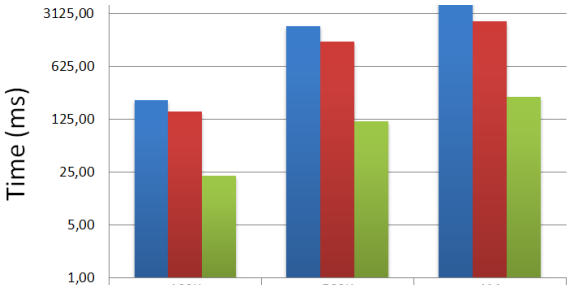
In the following, a comparison is made between our approach and existing and most used algorithms from the recent literature.

The approach is compared with the last releases of two other broad phase algorithms of the Bullet library [Bul], the 3D incremental sweep and prune (SAP) [CLMP95] (a 64 bits version has been developed to handle more objects) and the *btDbvt* algorithm which is based on dynamic AABB trees. Two different tests were performed. The first one is to determine the impact of the objects' movement on the computation time because many broad phase algorithms are optimized when many objects are static. Thus several performance measurements with varying percentage of moving objects are run. Results (cf. Figure 6) show that the approach is not impacted by the fact that the objects are static or dynamic. Unlike other approaches, whatever the percentage of moving object, our computation time remains constant. The second test is to assess the robustness of the algorithms according to an important number of objects. The algorithms is evaluated from one thousand to one million of objects, 100% moving. These tests were made with the same scenario where an object column of the same size falls on to a plan (cf. Figure 1) with the same Nvidia Quadro FX 3600M and a Intel Core2 CPU X7900 @ 2.8GHz. Results are illustrated on the Figure 5. The algorithm significantly outperforms the two existing methods with large-scale environments.

It is a bit difficult to compare results with the literature because there are only few approaches that have tried to compute the collision detection with more than a million of objects only using one GPU. It is also difficult to compare with other approaches because the GPU are different. There are several works studying the particles-based simulations with millions of elements but it goes away from our domain. The works of Tracy and al. [TBW09] focused on the improvement of the SaP on the CPU and their results present only environments with a low percentage of moving



	1K	5K	10K	50K
Bullet SAP	1,06	9,40	26,06	100,04
Bullet btDbvt	1,29	13,34	36,50	71,12
Our approach	1,35	1,39	2,41	10,21



	100K	500K	1M
Bullet SAP	223,86	2103,21	4621,78
Bullet btDbvt	156,31	1331,97	2439,31
Our approach	22,52	116,72	245,95

Figure 5: *Tests performed from one thousand to one million of moving objects with three different algorithms. All objects are the same size. The computation time remains interactive (<25ms) in an environment composed of 100,000 objects with a Nvidia Quadro FX 3600M.*

object. The unique result they indicate with 100% of moving objects consists of 3000 Objects. Their algorithm takes 4ms whereas ours takes 0,46ms (including transfer time). Liu and al. [LHLK10] have tested their approach on an environment consists of 960k spheres of three different sizes. However, they do not give the exact numerical values of the computation time of their algorithm, which makes the comparison difficult. The only comparable data are those from three measurements done on three different numbers of spheres: 64K, 128K and 256K. Their approach takes respectively 13.71ms, 48.16ms and 241ms to apply the algorithm during the broad phase stage. Our algorithm takes respectively 15.67ms, 30.68ms and 50.33ms for the same conditions of simulation. We are thus slightly below the performances for the environment of 64K spheres, 1.5 times faster for the one with 128K spheres and 5 times faster for the last one. Other approaches such as Pabst and al. [PKS10] and Lauterbach and al. [LMM10] are only tested on small *n*-body environments (300 Objects). The computation times of their algorithms of broad and narrow phase are not differentiated, only the total time is given. For a simulation of 300 spheres and 4 cones, the approach of Pabst takes 128.1ms to make the calculations in discrete mode and 184.8ms in continuous mode. It is always very difficult to estimate time made by the broad phase in an approach. Taking a broad phase which would represent 10%, 5%, 2% or 1 % of the total execution time, theirs would be respectively 12.81ms, 6.40ms, 2.56ms and 1.28ms. If we cross our results, our algorithm of SaP-Grid on GPU for the same type of environment takes exactly 0.41ms. Even for a broad phase which would represent only 1% of their total time, our algorithm would be three times faster. The additional difficulty of comparison is that Lauterbach et al. have carried out their tests on a Nvidia GTX 285 while Pabst et al. have performed on a Nvidia GTX 295 and we have done ours on a Nvidia Quadro FX 3600M which makes it very tricky to compare.

7. Conclusion and Future Work

A novel and efficient GPU solution has been presented to cull non-colliding object pairs. This new algorithm is based on a grid-based spatial technique and a new GPU mapping function that enables GPU threads to turn their ID into a unique objects pair without any global memory access. This paper brings new directions for the collision detection optimization. It has presented a different direction from recent collision detection works by going back to a semi brute-force approach. The novel approach enables to reduce the global memory accesses done by threads and leads to a faster execution time. To break definitively with the combinatorial of the algorithm, the approach is coupled with spatial subdivision. The kernel execution time is further reduced with the use of the square root approximation where a first characterization of the errors is presented. It enables the fixing of wrong computations leading to incorrect object pairs. The logical optimization with binary masks of the output data transfer has minimized the size and the time of the GPU-CPU transfer. Numerical and comparative results showed that this new algorithm enables to cull objects in less than 25ms with more than 100K moving objects on a Quadro FX 3600M. The comparisons made with the last releases of the well-known and most-used algorithms showed that this new model enables to further reduce the collision detection time. Compared to the literature, this model is, in most cases, as or faster than other ones.

There are many ways for future work. The GPU mapping function is fully scalable on multi-GPU platform so it could be interesting to evaluate its performance on bigger platform. The next step of our work will focus on the second stage (narrow phase) of the collision detection pipeline in order to know if it is possible to apply such a parallel spatial brute force approach. Using this approach not only on bounding volumes but also on triangles or polygons could be an interesting way. The grid establishment and update can be improved and better mapped on to the GPU architecture. A

study on the optimal number of cells into the grid is crucial and also on the cell size. It is now essential to take into account the run-time architecture to speed-up collision detection algorithms. We think that the key challenge is in the establishment of mapping models between the algorithms and the architectures.

8. Acknowledgments

The authors want to thank Alison Day and Colin Moore (Wechsler-Reya Lab) for their help in the English review process. This research was supported by Bretagne Region under project GriRV N^o4295.

References

[AGA11] AVRIL Q., GOURANTON V., ARNALDI B.: Dynamic adaptation of broad phase collision detection algorithms. In *VR Innovations (ISVRI) - IEEE International Symposium* (March 2011), pp. 41–47. 2

[Ber97] BERGEN G. V. D.: Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools 2*, 4 (1997), 1–13. 2

[BT95] BANDI S., THALMANN D.: An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies. *Comput. Graph. Forum 14*, 3 (1995), 259–270. 2

[Bul] BULLET: http://bulletphysics.org/. 8

[CLMP95] COHEN J. D., LIN M. C., MANOCHA D., PONAMGI M. K.: I-collide: An interactive and exact collision detection system for large-scale environments. In *SI3D* (1995), pp. 189–196, 218. 2, 8

[EG07] EITZ M., GU L.: Hierarchical spatial hashing for real-time collision detection. In *Shape Modeling International* (2007), IEEE Computer Society, pp. 61–70. 2

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the ACM Conference on Computer Graphics* (New York, Aug. 4–9 1996), ACM, pp. 171–180. 2

[GLM05] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Quick-cullide: fast inter- and intra-object collision culling using graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM, p. 218. 2

[GRLM03] GOVINDARAJU N. K., REDON S., LIN M. C., MANOCHA D.: Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), Doggett M., Heidrich W., Mark W., Schilling A., (Eds.), Eurographics Association, pp. 025–032. 2

[HRF09] HERMANN E., RAFFIN B., FAURE F.: Interactive physical simulation on multicore architectures. In *EGPGV* (2009), Debattista K., Weiskopf D., Comba J., (Eds.), Eurographics Association, pp. 1–8. 2

[Hub95] HUBBARD P. M.: Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics 1*, 3 (1995), 218–230. ISSN 1077-2626. 2

[KHeY08] KIM D., HEO J.-P., EUI YOON S.: *PCCD: Parallel Continuous Collision Detection*. Tech. rep., Dept. of CS, KAIST, 2008. 2

[KHH*09] KIM D., HEO J.-P., HUH J., KIM J., YOON S.-E.: HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs. *Comput. Graph. Forum 28*, 7 (2009), 1791–1800. 3, 7

[KP03] KNOTT D., PAI D. K.: Cinder: Collision and interference detection in real-time using graphics hardware. In *Graphics Interface* (2003), pp. 73–80. 2

[LCF05] LUQUE R. G., COMBA J. L. D., FREITAS C. M. D. S.: Broad-phase collision detection using semi-adjusting BSP-trees. In *SI3D* (2005), Lastra A., Olano M., Luebke D. P., Pfister H., (Eds.), ACM, pp. 179–186. 2

[LG98] LIN M. C., GOTTSCHALK S.: Collision detection between geometric models: a survey. In *Proceedings of the 8th IMA Conference on the Mathematics of Surfaces (IMA-98)* (Winchester, UK, Sept. 1998), Cripps R., (Ed.), vol. VIII of *Mathematics of Surfaces*, Information Geometers, pp. 37–56. 2

[LG07] LE GRAND S.: Broad-phase collision detection with cuda. *GPU Gems 3 - Nvidia Corporation* (2007). 2

[LHLK10] LIU F., HARADA T., LEE Y., KIM Y. J.: Real-time collision culling of a million bodies on graphics processing units. *ACM Trans. Graph 29*, 6 (2010), 154. 2, 7, 9

[LMM10] LAUTERBACH C., MO Q., MANOCHA D.: gproximity: Hierarchical gpu-based operations for collision and distance queries. In *Computer Graphics Forum (EUROGRAPHICS Proceedings)* (June 2010), vol. 29, pp. 419–428. 3, 9

[Lom03] LOMONT C.: *Fast Inverse Square Root*. Tech. rep., Dpt of Mathematics - Purdue University, Indiana, 2003. 5

[McE07] MCENIRY C.: *The Mathematics Behind the Fast Inverse Square Root Function Code*. Tech. rep., 2007. 5

[Mir97] MIRTICH B.: *Efficient Algorithms for Two-Phase Collision Detection*. Tech. rep., Mitsubishi Electric Research Laboratories, Dec. 1997. 2

[Ove92] OVERMARS M. H.: Point location in fat subdivisions. *IPL: Information Processing Letters 44* (1992). 2

[PKS10] PABST S., KOCH A., STRAβER W.: Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. In *Comput. Graph. Forum* (2010), vol. 29, pp. 1605–16212. 3, 7, 9

[TBW09] TRACY D. J., BUSS S. R., WOODS B. M.: Efficient large-scale sweep and prune methods with AABB insertion and removal. In *VR* (2009), IEEE, pp. 191–198. 2, 8

[Ter07] TERDIMAN P.: Sweep-and-prune: Multi-sap / http://www.codercorner.com/sap.pdf, 2007. 7

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Comput. Graph. Forum 24*, 1 (2005), 61–81. 2

[TLW11] TANG C., LI S., WANG G.: Fast continuous collision detection using parallel filter in subspace. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 71–80. 2

[TMLT11] TANG M., MANOCHA D., LIN J., TONG R.: Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games* (2011), I3D '11, ACM, pp. 63–70. 3

[TMT09] TANG M., MANOCHA D., TONG R.: Multi-core collision detection between deformable models. In *Symposium on Solid and Physical Modeling* (2009), Bronsvoort W. F., Gonsor D., Regli W. C., Grandine T. A., Vandenbrande J. H., Gravesen J., Keyser J., (Eds.), ACM, pp. 355–360. 2

[TPB08] THOMASZEWSKI B., PABST S., BLOCHINGER W.: Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics 32*, 1 (2008), 25–40. 2